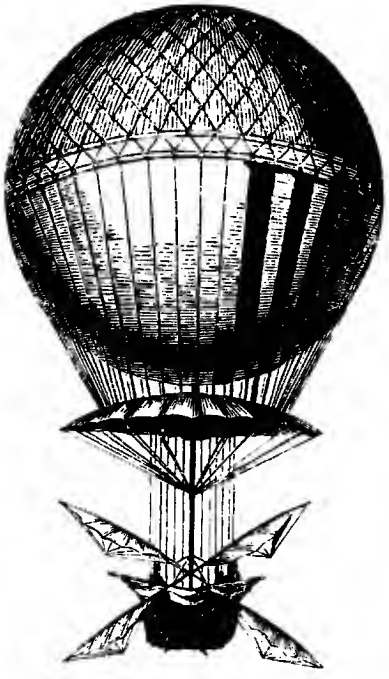
PILOT
User's Guide

MORROW

PILOT
User's Guide

MORROW
600 McCormick Street, San Leandro, California 94577



MORROW, INC.
600 McCormick Street
San Leandro, California 94577

Produced by William Harris and John VanderWood.

PILOT User's Guide Contents

1 Introduction

History of PILOT

THE PILOT (Programmed Inquiry, Learning, or Teaching) programming language was developed at the University of California at San Francisco in the early 1970's. It was originally designed to enable teachers to easily program custom-made lessons for their classrooms. As such, the language was kept as simple as possible, and the concept was developed that complete programs should be able to be written using only a very few "Core" instructions.

Morrow has adapted that old version of PILOT to work efficiently in today's environment. However, the concept of keeping PILOT as simple as possible has remained. As a result, you should be able to write a simple PILOT program in just a few minutes. Eventually, though, by increasing the level of complexity and the kinds of instructions in your program, you can expand PILOT's capabilities far beyond its original use as solely a teaching tool. You will, for example, be able to create instruction guides, self-teaching programs, and interactive menus.

If you've never done any programming before, PILOT is the right language for you. One thing that will make the learning easier is to have a project or goal that you want to accomplish before you begin. Then, as you proceed, you'll be able to fit the pieces of information into the bigger picture of your project. That way, when you encounter a new instruction, instead of saying "Oh no, more details!" you'll say "Aha! Just what I've been looking for!"

How to Use This Manual

This is how the PILOT manual is organized:

Section 2 gives a brief overview of the entire process of writing and running a PILOT program, and describes how to make a blank diskette with a copy of PILOT.

Section 3 introduces you to some of the "core instructions."

Section 4 has you writing and running actual PILOT programs.

Section 5 shows an example of a complex, functioning PILOT program.

Section 6 gives some hints about "common answers," and thoroughly discusses matching.

Section 7 is an alphabetical listing of all PILOT instructions, to be used for reference.

Section 8 covers the relatively unpleasant topic of error messages, and gives the steps to take to solve the problems.

The Appendix lists the syntax of every PILOT instruction.

2 Getting Started

Summary of the Procedure

IN ORDER TO use the PILOT program, you will first format a blank diskette, and then copy the PILOT interpreter program PILOT.COM onto that diskette. After those preliminaries, you will write PILOT programs by starting up NewWord or some other text editor and changing your logged disk drive to "b:" (where the PILOT.COM diskette should be). Then you'll type in your PILOT program just as if you were typing a letter, saving it on the PILOT.COM diskette.

After leaving the text editor, you will run a CP/M command consisting mainly of the word "PILOT" and the name of the command file you created. With a Micro Decision, you can do this from the Utility Menu. With other CP/M machines, you would do so from the "A>" or "B>" prompt.

Once you've entered the startup command, your program begins running, displaying those things on the screen that you've designed, while asking for users' responses.

Note to MD-11 owners: In general, you will be dealing with your A: (hard disk) drive exclusively, and no floppies will be involved. Those places where your procedure differs from the norm will be clearly pointed out.

If this whole business sounds complicated, we assure you, it isn't. You'll get step by step instructions and examples that will show you that it isn't hard at all.

NOTE: PILOT programs can be created on any text editor. For the purposes of this manual, we will assume you are working with NewWord files. If you are not, simply substitute your own text editor instructions for the NewWord instructions.

In addition, we are assuming that you are working through the menu system of the Morrow Micro Decision. If you are using some other CP/M computer, we trust you know enough about the basics of CP/M to be able to format diskettes and copy files.

Formatting Diskettes and Making a Copy of PILOT.COM

(This section does not pertain to the MD-11.) You will need a diskette to store your PILOT programs. First, with your CP/M system diskette in drive A: and a blank diskette in drive B:, prepare to format the blank diskette by selecting the Format option from the Utility Menu. Your responses to the prompts that are displayed should be those listed below.

1 Format a diskette

B Disk drive to be used

D Double sided format (MD-3 models only)

Then type the RETURN key.

When that is finished and the Utility Menu is back on the screen, type **C** to change the "currently logged drive". At the "Enter new drive" prompt, respond **A**.

Next, select the "Copy a file or files" option from the Utility Menu. Here are your responses:

N No, you don't want to copy all of the files on the current drive.

PILOT.COM A new file name, followed by RETURN.

B The drive you're copying to.

When the copy is finished a few seconds later, *you're done* with preparing your diskette. (The more sophisticated among you could now SYSGEN the diskette, eliminating the need for the CP/M system diskette; however, doing so is definitely not necessary if you aren't sure what this means.)

3 PILOT Instructions

A Point of Clarity

ONE thing you might want to get straight before we start discussing the makeup of PILOT programs is the difference between the PILOT ***interpreter*** program (PILOT.COM) and the PILOT programs you will be writing.

PILOT.COM is a program that you have to load into your computer before it can understand the PILOT instructions that you have written with your text editor. It reads your commands and sees to it that the terminal, computer, and disk drives respond as you intend them to. You will employ the same PILOT.COM interpreter program no matter what kind of PILOT program you come up with.

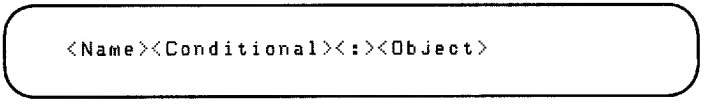
The Basic Form of Instructions

A PILOT program consists of a series of instructions which tell the computer exactly what to do. All of these instructions must follow a certain form to be understood by PILOT.COM. The form (or "syntax") of a PILOT instruction is basically composed of four different, important parts:

1. the name before the colon,
2. the "conditional,"
3. the colon itself, and
4. the object after the colon.

(One additional part, the "label," we'll introduce later).

In diagram form, then, a typical PILOT instruction looks like this:



The first part, the **name** of the instruction, is the part that tells the program what action to perform (or even to do nothing but wait). It is mandatory, of course, since the simpleminded computer is hopelessly lost without it.

The second part, the **conditional**, is optional, and is usually a "Y" or an "N" to indicate yes or no. The conditional tells the computer to do something *only if* certain things match up or if some other special conditions exist. Therefore an instruction *without* a conditional gets carried out no matter what; one *with* a conditional may or may not get done. This will become clearer later, when you see some examples of conditionals at work.

The third part, the **colon** (":") itself, is also required, since it separates the name of the instruction from the object of the instruction.

The fourth part of the instruction, the **object**, is sometimes optional and sometimes mandatory, depending on which instruction you're talking about. It tells the computer what to put on the screen, what to compare an answer with, or where to go next, to mention a few uses.

Now, we'll take a look at some of the PILOT core instructions that follow this pattern.

Some Core Instructions

As mentioned in the introduction, the original PILOT interpreter program only recognized a few Core instructions. These instructions still form the base for the PILOT interpreter you will be working with, and complete programs can be constructed using only some of the core instructions.

The core instructions we will be using to create our first sample PILOT program in the next section are these:

NOTE: Don't worry too much about the meanings. They will be discussed much more thoroughly in later sections. They're given here simply to introduce you to the kinds of things the first sample program in the next section will be doing.

Instruction	*Meaning*
T:	TYPE. Instructs the computer to display on the screen whatever follows the colon.
A:	ACCEPT. Tells the computer to wait for an answer from the user, and accept whatever the user types as the answer.
M:	MATCH. Instructs the computer to compare the answer the user gave to the answer following the colon.
TY:	TYPE IF YES. This TYPE statement, with the addition of the conditional Y, will display the text following the colon only if a certain condition exists, such as if the user's response matches the object of an M: instruction.
TN:	TYPE IF NO. This TYPE statement, with the addition of the conditional N, will display the text following the colon only if the special conditions don't exist.
END:	The last statement of every PILOT program. When running the program from the Micro Decision Utility Menu, END: means go back to the Utility Menu. On other systems it normally means go back to the CP/M prompt. You can use END: at other places than the end of a program, say, to allow a user to quit in the middle of some sequence.

Using only these statements, we can write our first simple PILOT program.

4 Creating Sample PILOT Programs

The Simplest Useful Program

IN its most basic form, a PILOT program asks a question, accepts and compares an answer, and does something based upon that answer. We're going to create an actual PILOT program that does just that.

Creating the Simplest Program

First, you have to start up your text editor. On the Micro Decision Main Menu, select the option for NewWord or a similar text editor. The system will instruct you to put your NewWord Working Diskette in Drive A. After you change diskettes, your system will automatically take you into NewWord when you press RETURN.

When you have reached NewWord's Opening Menu, insert your newly-formatted diskette with PILOT.COM on it into Drive B. Select L to change the logged disk drive to "b:". Then type N (*not "D"*) to create a new **non-document** file. (MD-11 owners should not change the logged drive. However, they will select a User Area, which is up to their discretion. They should, however, make a mental note of which User Area they've chosen.)

NOTE: It is important to type your program using the text editor's non-document mode ("N" on NewWord's Opening Menu, as mentioned above). Programs entered using the document mode ("D" on the Opening Menu) probably will not work correctly. What we're saying, to the technically inclined, is that you don't want your text editor to use the high-order bit.

Give the "non-document" the name **SIMPLEST.PIL**. That name will be the name of our sample PILOT program. The .PIL ending is not mandatory, but is useful to identify the file as a PILOT program on your directory listing. Also, a program ending in ".PIL" is a little more convenient to run, since your command line can read simply "PILOT SIMPLEST" instead of "PILOT SIMPLEST.PIL."

Now, we'll start to write the program. A couple of things to be aware of: PILOT instructions don't have to be lined up starting in the first column, and they don't have to be capitalized either. But for the sake of legibility, these are good habits to get into.

First, type the question you want the user to answer as follows:

```
T:    The term "PILOT", as used in this manual, means:
T:      1)  Airplane Jockey
T:      2)  A Computer Language
T:    Type the number of your choice and press RETURN
```

When you run this program, the "objects" of the T: instructions (the four lines of text) will be displayed on the screen. The T's and colons won't be.

Next, instruct the computer to wait for and accept an answer:

```
A:
```

This tells the computer to wait indefinitely for a response and a RETURN, and to keep track of the response for use in the next instruction.

Then tell the computer to compare (MATCH) the user's answer to the right answer:

```
M: 2
```

This MATCH instruction tells the computer to compare the answer the user gave to the answer given after the colon. In this case, the computer will check the user's answer to see if it is "2" or not.

Now comes the important part. You can instruct the computer to do two different things, based on whether the answer the user has given matches the object of the M: instruction. This is done by telling the computer to type one thing if the answer matches, as follows:

```
TY: Very good!  My, aren't you the bright one!
```

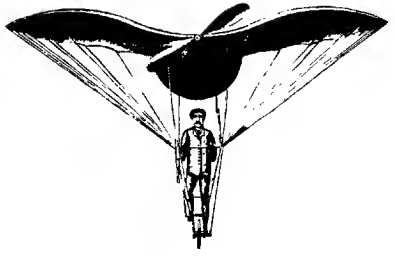
Or type a different response, if the answer doesn't match:

```
TN: Boy, are you a dunce!
```

The "Y" and the "N" following the T: instruction are called **conditionals**, and they are some of the decision makers of the PILOT program. Using them, the computer can look at a user's response and take action based upon that response.

Finally, after all of the other commands, add the END: statement to tell PILOT to quit and go back to the menu or prompt that you started it up from:

```
END:
```



Check to see that what you have typed looks exactly like this:

```
T:  The term "PILOT", as used in this manual, means:
T:       1)  Airplane Jockey
T:       2)  A Computer Language
T:   Type the number of your choice and press RETURN
A:
M:  2
TY: Very good!  My, aren't you the bright one!
TN: Boy, are you a dunce!
END:
```

You have now created an actual PILOT program, and your next step is to run it, i.e., get the computer to go through the program and ask the question you have written. To do this, save the program on the PILOT.COM diskette and get out of the text editor (^KX if you're using NewWord).

Running the Simplest Program

Let's begin by assuming you're looking at the Micro Decision Main Menu. Your CP/M system diskette is in Drive A and your PILOT.COM diskette that has SIMPLEST.PIL stored on it is in Drive B. (MD-11 owners: ignore directions on diskette placement.)

From the Main Menu, proceed to the Utility Menu. Select the "Execute a CP/M command" option.

At the "Command?" prompt, enter

```
B:PILOT SIMPLEST.PIL
```

and press RETURN. (MD-11 owners should not type the "B". Instead, they should type USER #. where "#" is the User Area that contains SIMPLEST.PIL. Then they should pick "Execute a CP/M command" a second time, typing the command above without the B:.)

If you're using a different CP/M computer, just enter this same command at the A> or B> prompt.

The PILOT sign-on message will appear and then the first "T:" statements of your program will be displayed:

```
The term "PILOT", as used in this manual, means:
     1)  Airplane Jockey
     2)  A Computer Language
Type the number of your choice and press RETURN
```

Type in "2" and press RETURN. The computer will congratulate you for your erudition and end the PILOT program, since you only asked the one question in this sample.

Next, to prove that the program will give a different response based upon a wrong answer, run the program again, but this time respond with a "1" to the question instead of a "2". When the computer chides you for being a dunce, remind it that it was able to do so only because you were smart enough to tell it how.

So you can see that you've created a computer program that will ask a question and give a different response based on the answer. The simple example we just wrote, of course, isn't very useful. However, by stringing several questions together, and providing more elaborate routines based upon a user's response, you can write much more complex programs.

The Next Steps

The last example demonstrates that a PILOT program can react in different ways to different responses. Our next program will start itself over again if the user asks it to, or skip information if the user wishes, by using a core instruction named "J:" (JUMP), and an additional part of a PILOT instruction called a label. After that, we'll write a program to produce a standard error message if the user makes a mistake, and another program to calculate a net price, by continuing to use labels, and adding a core instruction named "U:" (USE).

Creating a Program With a Label and JUMP

First follow the instructions for using your text editor, as outlined above for the SIMPLEST program, and start a new non-document file called **NEXT.PIL**.

Okay. Let's write some lines to ask a question, just like we did in the previous example, as follows:

```
T:  Would you like to:
T:    A)  Review Core Instructions from last example?
T:    B)  Look at the new instructions from this example?
T:  Type the letter you want and press RETURN
A:
```

You can see that so far this example follows the same principles as the last one, namely, a multiple choice question with branches in the action based on the answer given to the question.

First you need to pick which response to perform the MATCH test on.

NOTE: It doesn't matter which answer you choose as the answer to match. As you'll see when you work through the example, it just reverses the branching in a commutative way. In other words, if in our first example we had matched against the "1" response instead of "2", we could have kept the logic intact by also reversing the "TY:" and "TN:" responses. Confused? Don't worry; just type what we show you and you'll absorb the finer points as you go along.

Also note that the MATCH statement below will accept either a lower or upper case A as a match.

```
M:  A
```

So, if users type "A", they want to look at the instructions we used last time. Now we'll write the TYPE IF YES instruction which will allow them to see those instructions:

```
TY:  The core instructions we used for
TY:   the last example were "T:" (TYPE),
TY:   "A:" (ACCEPT), "M:" (MATCH)", and "END:".
```

Next, write the TYPE IF NO instruction that allows the user to look at the more advanced core instructions we'll soon be using in this example, as follows:

```
TN:  The new core instructions we are using in this
TN:   example are "J:" (JUMP) and "U:" (USE).
```

Fine. Now, if we didn't add anything else, you'd have a program essentially just like the last one we wrote. That is, it would ask the question, act on the response, and quit. However, by using another core instruction called JUMP, we can go back through the program and look at the other answer without having to run the program again. First, write an instruction to ask the users if they wish to go through the program again, as follows:

```
T:  Do you wish to see the other set
T:   of instructions? YES or NO
A:
M:  YES
```

So, you've asked the users if they wish to see the other set. Now, if they respond "yes," they can go back to the

beginning of the program and ask the original question again, using a JUMP statement:

```
JY:  *START
```

This tells the computer to jump to the **label** *START. We haven't talked much about labels yet, but they're important. A label provides a name for a given section of a program. By using a particular label, you can direct the program to go directly to that section of the program, rather than simply following down the list of instructions.

Labels can be made up of any combination of numbers and letters, but they must start with an asterisk, and they must either be the first statement on a line, or on a line by themselves.

So far, we've told the computer to JUMP to a label that doesn't yet exist. So you should go back to the beginning of our NEXT.PIL program and add the label, as follows:

```
*START
T:  Would you like to:
T:    A)  Review Core Instructions from last example?
T:              (etc.)
```

NOTE: Remember, the label you use this time must be *exactly* the same as the one you gave the computer in the JUMP statement. For example, we could also have told the computer to JY: *BEGIN, but then we would have had to write the label at the beginning of the program as *BEGIN.

Now, if the user answers "yes" to the question about seeing the other set of core instructions, the program will automatically jump back to the beginning and ask the question over again.

What if the user doesn't want to see the other set? This issue brings us back to the END: instruction:

```
JY:  *START
END:
```

So you see you didn't have to cover the possibility of the user answering "NO" by having a JUMP IF NO or TYPE IF NO or any such conditional. While these avenues are certainly open to you, this example shows that PILOT will simply ignore the JY: command if the match test failed, and go on to the next instruction. In this case, that is the END: statement.

Now let's check your program. Does it look like this?

```
*START
T:  Would you like to:
T:      A)  Review Core Instructions from last example?
T:      B)  Look at the Instructions for this example?
T:  Type the letter you want and press RETURN.
A:
M: A
TY:   The core instructions we used for the last
TY:    example were "T:" (TYPE), "A:" (ACCEPT),
TY:     "M:" (MATCH), and "END:".
TN:   The new core instructions we are using in this
TN:     example are "J:" (JUMP) and "U:" (USE).
T:  Do you wish to see the other set
T:   of instructions?  YES or NO
A:
M:  YES
JY:  *START
END:
```

You have now written your first program with a JUMP, that takes you back through the program to ask the same question over again, instead of simply ending after asking a question.

Now, run this program following the instructions outlined for program SIMPLEST.PIL. Be sure to choose all of the different options involved, and you will verify that the program does what we want it to do.

* * *

Now that you have finished verifying that the program works, you've seen that a JUMP statement may be used to go back and repeat a sequence in the program. To make things really fancy, we're going to add another JUMP statement, one that jumps us "forward" (i.e., skips over a section that the user doesn't wish to see), so that you can see a different way the flow of the program can be controlled.

Return to the NEXT.PIL program, using your text editor, and get ready to modify it.

Let's assume that the user has finished with the question about the core instructions, and has typed "NO". This means he doesn't want to "loop" back through the question again. With the way the program is now written, that means END.

First, delete the END: instruction so the program doesn't end at this point. You're going to add another question about PILOT, as shown:

```
T:  Would you like to review the history of PILOT?
T:     YES or NO
A:
M:  YES
```

If the user types "YES," he does want a review, so we'll add:

```
T:  PILOT is a programming language developed at
T:  UCSF to aid in constructing Computer Assisted
T:  Instruction materials.
```

However, if the user types "NO," he doesn't want to see that information, so we'll instruct the computer to skip the information by inserting a JN: (JUMP IF NO) instruction before the T: instruction:

```
M:  YES
JN:  *FORWARD
T:    PILOT is a programming language developed at
T:    UCSF (etc.).....
```

Now, we have to use the "*FORWARD" label, to give the computer a place to jump to:

```
*FORWARD
```

After this label, you can END or continue as you wish. The information about the history of PILOT has been skipped.

Let's finish this program by asking the user once more if he might find a review of the information to be entertaining:

```
T:  Would you like to go through these questions
T:    again?  YES or NO
A:
M:  YES
```

JUMP back to the start if the user wants to, as follows:

```
JY:  *START
```

We have already placed the *START label in the program, so we don't need to do that now. So, END the program with an END: statement. Check to see that the program looks like this:

```
*START
T:  Would you like to:
T:  A)  Review Core Instructions from last example?
T:  B)  Look at the instructions for this example?
T:  Type the letter you want and press RETURN.
A:
M:  A
TY:  The core instructions we used for
TY:   the last example were "T:" (TYPE),
TY:   "A:" (ACCEPT), "M:" (MATCH), and "END:".
TN:  The new core instructions we are using in this
TN:   example are "J:" (JUMP) and "U:" (USE).
T:  Do you wish to see the other set
T:   of instructions? YES or NO
A:
M:  YES
JY:  *START
T:  Would you like to review the history of PILOT?
T:     YES or NO
A:
M:  YES
JN:  *FORWARD
T:  PILOT is a programming language developed at
T:  UCSF to aid in constructing Computer Assisted
T:  Instruction materials.
*FORWARD
T:  Would you like to go through this lesson
T:    again? YES or NO
A:
M:  YES
JY:  *START
END:
```

(The blank line in the program is there to make it more readable. Blank lines, even several in a row, don't break the flow of the program in any way.)

Save the program on the same diskette as PILOT.COM and SIMPLEST.PIL. Then run it (following the instructions given for SIMPLEST.PIL) giving various responses, and you'll see how the JUMP statement works. Notice that the information about PILOT and UCSF would never appear if the right sequence of answers was given, i.e., the JUMP instruction can cause the program to simply skip those statements.

Adding USE Instructions—Creating Subroutines

You could continue adding more JUMP statements to the program, but they would still function in the same way, that is, go to a label and start going downward through the program from there. However, as your programs increase in complexity, you will often find it useful to go to a certain place in the program, do the instructions contained there, and then return to the spot in the main flow of the program from where you left. We'll now continue with the next PILOT core instruction, one called "U:" (USE), which allows us to go to a set of instructions called a **subroutine**, use those instructions, and jump *back* to where we came from.

To start, let's create a program called **USE.PIL**, following the instructions for SIMPLEST.PIL to set up the text editor.

This program will be for a menu system, where clerks would be choosing different options. If the clerks select an option that does not exist, or they mistype their selection, we want to ignore their entry and tell them to retype the answer.

Now, how can the "U:" (USE) instruction help us? For one thing, it lets you write instructions only once for a situation that will occur over and over again. For example, in this sample program, we'll assume that we have several places in the program where we wish to give users a standard message if they do not enter any of the single digits 1, 2, or 3 in response to a multiple choice among those numbers. We could write statements into the program at each place where the error message would occur, but that's making work for ourselves. It's easier to place the statements that give the message in one place in the program, and go use those statements each time we need them. This is called using a subroutine. The error message section of the program is the soubroutine.

Let's begin as follows:

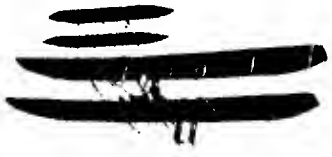
```
*START

*WHICHMENU
T:  Select one of the following options:
T:       1)  Review Company Policy on this
T:                   type of Sale
T:       2)  Review Current Inventory
T:       3)  Review Customer Status
T:    by typing its number and pressing RETURN.
A:
```

If the clerks were unfamiliar with the conventions of exact data entry, they might type the name of the number instead of its digit, e.g, "ONE" instead of "1," in which case the program would not work if we wrote it so that it needed the single-digit number. (There's a couple of ways to get around the exact match principle. However, we'll leave that discussion until later, and in any case the principle of what to do in case of no match remains the same.)

This type of understandable human error can quickly lead to computer-frustration, so we want to indicate to the clerk exactly how the answer must be formatted. Furthermore, given the vagaries of clerks, it's possible that they might answer this question with a single digit, but later in the program answer a similar three-choice request with a "ONE," or "#1". So, whenever the three choices exist, and the clerks make a mistake, or simply mistype an answer, we want to inform them of the error.

First we'll MATCH the digits, to set up the program's reactions to correct entries:



NOTE: MATCH instructions can be tricky sometimes, and we'll postpone discussing them for the moment. For now, please be sure that the following instruction is typed exactly as follows, i.e., space-1-space (comma), space-2-space (comma), space-3-space.

```
M:  1 , 2 , 3
```

If a clerk types one of the three digits, the MATCH will occur and we'll instruct the program to continue by JUMP-ing to another MATCH instruction:

```
JY:  *MATCHACCEPT1
```

We'll write the instructions that go with the label *MATCHACCEPT1 in a moment, but first we'll instruct the computer to USE the subroutine *ERROR3 if the clerk doesn't type one of the three digits (USE IF NO):

```
UN:  *ERROR3
```

Let's write that subroutine now:

```
*ERROR3
T:   I'm sorry.  I need the number 1, 2, or 3.
T:   Please reread the question and type just
T:   the single digit.
```

A wrong entry will now take us to the subroutine and type the statement on the screen. Now, we need to signal the computer to end the subroutine (but NOT the whole program) and return to the main program. This is done by using another core instruction, the "E:" instruction. Simply type:

```
E:
```

NOTE: Do not type "END:". That would end the whole program, instead of just the subroutine.

The E: instruction, when used with a subroutine called by a U: (USE) instruction, will automatically return us to the next instruction in the program following the U: instruction that led to it. The same is true of UN: and UY: instructions, too.

In this case, we want to ask the question again, so we'll write a jump statement to return to the start of the question:

```
J:  *WHICHMENU
```

and then we could continue with the main program, in this case by defining what to do for each of the three choices, e.g.:

```
*MATCHACCEPT1
M:  1
JY:  *POLICY
JN:  *MATCHACCEPT2
*POLICY
T:  What Kind of Sale is it?
T:       1)  Equipment
T:       2)  Software
T:       3)  Documentation
A:
```

Once again you can USE the subroutine *ERROR3 that we have already written, if the user doesn't answer the type-of-sale prompt correctly:

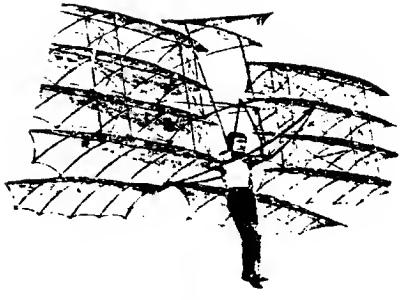
```
M:  1 , 2 , 3
JY:  *SALETYPE
UN:  *ERROR3
```

If the user answered correctly with one of the three digits, the program would jump to the label *SALETYPE, and continue down the program from there, thus skipping the UN: instruction and the subroutine. If they answered incorrectly, the *ERROR3 subroutine would print our standard message, and then would return us to the *WHICHMENU label at the beginning of the main program.

NOTE: Just to clarify things for you a little, if you are wondering what's the difference between a U: with subroutine and a J:, this is it: You could use two JUMPS to display an error message and then go back to the prompt. However, this method only allows you to go back to ONE SPECIFIC place, because with a jump you would need to name a label to return to. A U: and subroutine are far more flexible since they let you go back to wherever you came from without naming a certain place in the program.

Okay. After returning to the main program from the subroutine, we would want to jump back again to the question, as follows:

```
J:  *POLICY
```



Now you need to write a section labeled *SALETYPE:

```
*SALETYPE
M:  1
TY:  Charge a markup of 25% on equipment.
M:  2
TY:  Charge a markup of 15% on software.
M:  3
TY:  We do not sell documentation only.
J:  *MOREQUESTIONS
```

This section will give us the policy answers, and then jump us to a *MOREQUESTIONS section which we will place at the end of the program.

Just to finish the program quickly, let's write our options 2 and 3 as follows:

```
*MATCHACCEPT2
M:  2
TY:  Sorry.  The Current Inventory part of this program
TY:  has not yet been written.
JN:  *MATCHACCEPT3
J:  *MOREQUESTIONS

*MATCHACCEPT3
T:  Sorry.  The Customer Status part of this program
T:  has not yet been written.
J:  *MOREQUESTIONS
```

Now, we'll end it by adding the standard question, as follows:

```
*MOREQUESTIONS
T:  Do you need to review other Company Policy,
T:  Current Inventory, or Customer Status?
A:
M: YES
JY:  *START
ENO:
```

Check your program to see that it looks like this:

```
*START

*WHICHMENU
T:  Select one of the following options:
T:       1)  Review Company Policy on this
T:                   type of Sale
T:       2)  Review Current Inventory
T:       3)  Review Customer Status
T:     by typing its number and pressing RETURN.
A:
M:  1 , 2 , 3
JY:  MATCHACCEPT1
UN:  *ERROR3
J:  *WHICHMENU

*ERROR3
T:   I'm sorry.  I need the number 1, 2, or
T:   3.  Please type just the single digit.
E:

*MATCHACCEPT1
M:  1
JY:  *POLICY
JN:  *MATCHACCEPT2
```

```
*POLICY
T:  What kind of Sale is it?
T:       1)  Equipment
T:       2)  Software
T:       3)  Documentation
A:
M:  1 , 2 , 3
JY:  *SALETYPE
UN:  *ERROR3
J:  *POLICY
*SALETYPE
M:  1
TY:  Charge a markup of 25% on equipment.
M:  2
TY:  Charge a markup of 15% on software.
M:  3
TY:  We do not sell documentation only.
J:  *MOREQUESTIONS*MATCHACCEPT2
M:  2
TY:  Sorry.  The Current Inventory part of this program
TY:  has not yet been written.
JN:  *MATCHACCEPT3
J:  *MOREQUESTIONS

*MATCHACCEPT3
T:  Sorry.  The Customer Status part of this program
T:  has not yet been written.
J:  *MOREQUESTIONS

*MOREQUESTIONS
T:  Do you need to review other Company Policy,
T:  Current Inventory, or Customer Status?
A:
M:  Yes
JY:  *START
END:
```

Save and run the program, and you'll see how the error message appears appropriately.

Now you have seen one way to use a U: instruction. When common information is to be used over and over again, a subroutine can be extremely helpful.

USE—A Subroutine With a COMPUTE Instruction

We'll write one more program using a U: (USE) instruction, but this time we'll also introduce the "C:" (COMPUTE) instruction.

The C: (COMPUTE) instruction gives us a way to deal with numbers. Using C: you we can make your program add or subtract sums, or count right answers, or figure sales totals.

NOTE: There are some limitations on the COMPUTE instruction. It can only **add** or **subtract** whole numbers whose results are in the range 32,567 and −32,568. Also, do not use commas within numbers in C: commands.

For this example, let's assume that a clerk wants to know what to charge for a given part, and that a surcharge on the part is necessary depending on which division of the factory it comes from. Name the program **COMPUTE.PIL**, and start as follows:

```
*START
T:  What is the cost of the product?
A:  #PRODUCTCOST
```

The A: instruction, when used as above, has stored the cost of the product as entered by the user under the name #PRODUCTCOST, which later can be used in the computation. Numbers stored under names with a "#" sign in front of them are called **numeric variables**, and may be used in various ways in COMPUTE statements. We'll use two of them in this program, and others will be explained more thoroughly in the reference section under C: COMPUTE.

For now, let's continue with another question and answer to find out which division the product came from:

```
T:  Which division is the product from?
T:     A)  Oakland
T:     B)  New York
T:  Type the letter of your choice and press RETURN
A:
M:  B
UY:  *NYCHARGE
```

This part of the program, like the programs we have already written, will accept the user's response and, if the response is "B", will go to the section of the program labeled "*NYCHARGE", use the instructions there, and then jump back to the next instruction in the main program.

Now, we'll write the *NYCHARGE section of the program:

```
*NYCHARGE
T:  Will there be additional shipping charges?
T:  If so, how much (Enter 0 if none)
A:  #SHIP
C:  SURCHARGE = SHIP + 10
E:
```

After the initial question, and the storage of the response in the numeric variable #SHIP, we have written our first example of the COMPUTE instruction. The program will use the value of the number stored in the numeric variable #SHIP, add 10 to that number, and store the result in the numeric variable SURCHARGE (when numeric variables are used in COMPUTE instructions, they do not have to have the # sign in front of them).

After the computer has done that computation, it will end the subroutine and return to the main program.

Continuing with the main program:

```
C:  PRICE = PRODUCTCOST + SURCHARGE
T:  The price to charge for the product is
T:    #PRICE dollars.
```

Here's our second example of a COMPUTE instruction. We have just told the computer to add together the numbers stored in PRODUCTCOST and SURCHARGE (rembember, we just computed the SURCHARGE in the *NYCHARGE subroutine), and store the result in the numeric variable PRICE. Then, in our TYPE statement, we told the computer to print the value of the numeric variable #PRICE. Note the use of the "#" sign. In any other instruction than a COMPUTE statement, the # sign is necessary.

Special note on special characters: $,

If we wanted to print a "$" in front of the PRICE in the above example, we would have to place *two* dollar signs in front of #PRICE in the T: command ($$#PRICE, in other words). The PILOT program uses a single dollar sign for a special purpose (string variables; we'll talk about them soon), so we have to put two signs in the program in order to say that this is not a special symbol, it's really a dollar sign. Similarly, if you want to print a pound sign (#) to indicate a quantity, say, you would need to put two of them in front of a variable contained within a TYPE command. In summary, if you want to display the line "How to Make $$$" your instruction would be:

```
T: How to Make $$$$$$
```

We've written a program that will compute and display a price, based upon a C: statement and some numeric variables. Now, in our standard question asking if the user wants to ask for more prices, we'll introduce two more wrinkles: a **DEF:** (DEFINE) instruction and **string variables**.

```
DEF:  $YES  yes , y
T:  Do you need to ask about another price?
A:
M:  $YES
```

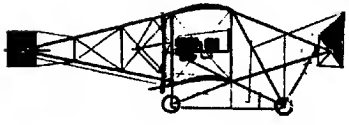
The DEF: instruction above indicates to the computer that either a "yes" or a "y" will be accepted as the value of the string variable $YES. String variables begin with a "$", and may contain words or individual characters. They can be used as above, to allow multiple answers, or they may save a user's name to be used later to give the impression that the computer is talking to the user, etc. In this case, if the user types either a "y" or "yes," the computer will consider it as a match for the string variable $YES.

So, if users type a "y" or "yes," they do wish to find another price, and we'll jump back to the start. If they type any other response, we'll assume we should end the program:

```
JY:  *START
END:
```

We still have a few instructions left to write. If the program jumps back to the start, we will be using the numeric variables over again to compute some more prices. When a numeric variable is used, the value that is assigned to it remains there until we do something about it, i.e., either assign it another value or reset all numeric variables to zero.

In our example, the old shipping charges and surcharges would remain as the variables and be added to any new costs. For example, if we had first computed a New York price, the old value for #SURCHARGE would still be there when we came back through the program again. If we then asked for an Oakland price, the program would add the value left in #SURCHARGE to the cost of the product, and thus give us a wrong answer. Therefore, we must make sure the value of the variable #SURCHARGE (and all other variables) is equal to zero. There are two alternate ways to do this:



1) If you're using a PILOT version *later* than Version 5.1, the instruction "RESET:" may be used. This instruction sets the value of all numeric variables to zero.
2) If you're using PILOT version 5.1 or earlier, then you must reset each numeric variable separately by computing its value as equal to zero.

So, in this program, we're going to use several C: (COMPUTE) instructions. If you can use RESET:, one RESET: instruction would take the place of *all* of the C: instructions we're going to write.

Since we want to be certain that all variables are correct when we recompute them, we're going to insert the C: instructions right after the start:

```
*START
C:  PRODUCTCOST = 0
C:  SURCHARGE = 0
C:  SHIP = 0
C:  PRICE = 0
T:  What is the cost of the product?
T:  (etc.)
```

All of the numeric variables in our program have now been set to zero, and will be set to zero each time you start again. The prices will thus be computed correctly each time.

Check your program to see if it looks like this:

```
*START
C:  PRODUCTCOST = 0
C:  SURCHARGE = 0
C:  SHIP = 0
C:  PRICE = 0
T:  What is the cost of the product?
A:  #PRODUCTCOST
T:  Which division is the product from?
T:     A)  Oakland
T:     B)  New York
T:  Type the letter of your choice and press
T:    RETURN
A:
M:  B
UY:  *NYCHARGE
C:  PRICE = PRODUCTCOST + SURCHARGE
T:  The price to charge for the product is
T:    #PRICE dollars.

DEF:  $YES yes,y
T:  Do you need to ask about another price?
A:
M:  $YES
JY:  *START
END:

*NYCHARGE
T:  Will there be additional shipping charges?
T:  If so, how much (For none, enter 0)
A:  #SHIP
C:  SURCHARGE = SHIP + 10
E:
```

(or simply RESET: for appropriate versions)

Note that the subroutine has been placed at the end of the program. Often, this is one good way of keeping your program organized.

Now, save and run the program to verify that it works. Choose the different New York and Oakland options, and use different prices, and you will see how the price changes correspondingly. Also, type in both "yes" and "y" in response to the question about more prices, and you will see that both of them work.

You now have the basic tools to make the PILOT language work for you. There are some other commands that are useful (we'll talk about some that make your programs prettier in the next section), but you have learned the basics of the system. Unfortunately, no manual on how to write computer programs (and that is what you have been doing) can teach you all that you need to know. It is necessary, and a lot more fun, to learn to program by writing programs. Experiment, and play, with the PILOT language, and you will find that soon you will be able to write useful programs very quickly.

5 A More Advanced PILOT Program

UP TO NOW, we have been working with small PILOT programs that we have written ourselves. Now, it's time to take a look at a functioning PILOT program that someone else has written.

The programs which control the menus for your Micro Decision are written in PILOT. We're going to look at excerpts from the main menu-controlling program, which is called MICRO.PIL. This program contains several additional PILOT instructions which we haven't discussed yet. In particular, there are several PILOT instructions which help make either the program listing itself or the display on the console more readable.

Instructions That Make a PILOT Program Look Nice—R:, LF:, CLRS:, CUR:, TNR:, and Braces

First, there is an instruction called **R: (REMARK)** which allows the PILOT programmer to put remarks into the PILOT program which will not appear on the console when the program is run. Such remarks are useful for indicating the different sections of the program. For example, you could write:

```
R:  This section displays the Welcome Screen
*START
T:  Welcome to Pilot!  We hope you enjoy using (etc.)
```

The statement following the colon in the R instruction would *not* display on the screen, but would display when you looked at the program. You will see other examples of R: instructions in the upcoming example.

There are several instructions which make the display of a PILOT program on the screen more pleasant to view. You may have noticed a tendency for the screen to look a little crowded when our programs were run. Three instructions which can counteract that crowdedness are "LF:" (LINE FEED), "CLRS:" (CLEAR SCREEN), and "CUR:" (SET CURSOR).

LF: (LINE FEED) simply instructs the computer to move down the screen the indicated number of lines, leaving those lines blank. You can tell the computer to move down as many lines as you want, but remember that a value greater than the number of lines on your screen will roll any old information off the top of your screen.

For example, if you wanted 2 blank lines between a user's response and the next question, you could write:

```
TY:  That's a correct answer.
LF:  2
T:   What does the PILOT instruction "LF:" do?
```

Putting adequate line feeds into a screen display helps to make it easier to read.

NOTE: A T: (TYPE) instruction, with no characters after the colon, will also insert one blank line per instruction.

The **CLRS: (CLEAR SCREEN)** instruction simply tells the computer to blank the screen and put the cursor in the upper left hand corner. This is useful when going to a completely new subsection of the program, or when starting a series of questions over. No characters are typed after the colon, so the command is simply:

```
CLRS:
```

The next instruction, the **CUR: (SET CURSOR)** command, will place the cursor on the indicated column and line of the screen. This command is useful when you want the user to type his/her information in a particular place on the screen. After the colon, write the column number first, so that if you wanted the cursor to appear in the fourth column and twelfth line of the screen, you would write:

```
CUR:  3,11
```

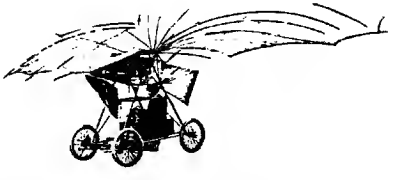
This is because the first line and column of the screen are considered to have the number zero instead of one.

The **TNR: (TYPE WITH NO RETURN)** instruction allows a response to be typed on the same line as a question. For example:

```
TNR:  Which option do you want?  Option **
```

This would allow the option number to be entered directly after the "#" sign. Note that "#" is a special character, thus two in a row are required for one to be displayed.

Finally, if your particular terminal can "highlight" characters (i.e., cause them to appear brighter or dimmer), there are two special characters used in the following example which will also change the display. A **left brace** ({), when written after the colon in a T: instruction, will be translated into a blank, but anything following it will be highlighted. Similarly, a **right brace** (}), when written after the colon in a T: instruction, will return the screen to normal intensity. Thus, you can highlight special words on the screen display.



A Working Example—Part of the Program MICRO.PIL

Now we can examine MICRO.PIL. There will still be instructions you don't understand. You can look them up in Section 7 if you want, but don't worry too much about them. You should be able to follow much of the main flow of the program.

Here's the first few lines we'll discuss from the MICRO.PIL program:

```
R:  display main menu
*BEGIN
ESC:*EXITMENU
U: *MENU1
```

As you can see, the programmer started off this section with a remark to indicate that the next few lines would display the main menu. Then he labeled the beginning of the program with the label *BEGIN, to give a place to go back to. The next command, "ESC:", we haven't talked about yet, but simply gives a little routine (which would appear after the label *EXITMENU further on in the program) to do in case the ESCAPE key is pressed. Refer to the detailed explanations of each instruction for further information.

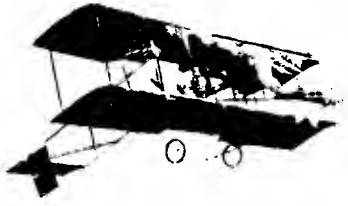
Then, the programmer has written the first USE: instruction, telling the program to go to the subroutine *MENU1, and perform the instructions there. Let's display that subroutine (it is actually located quite a bit later down in the program). That subroutine is:

```
R: = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
R:
R:  Menu displays
R:
*MENU1
CLRS:
T:                              {M A I N   M E N U}
T:  This menu is your road map through the CP/M operating system. To
T:  perform these functions, just enter the appropriate number after
T:  the prompt below, then follow the instructions given.
LF:1
T:        {1      NewWord}                  Word Processing
T:        {2      SuperCalc}                Financial Analysis
T:        {3      Correct-It}               Spelling Checker / Corrector
T:        {4      Personal Pearl}           Data Base Manager
T:        {5      Quest}                    Bookkeeper System
T:        {6      MBASIC-80}                Microsoft BASIC
T:        {7      BaZic}                    North Star Compatible BASIC
T:        {8      CP/M Tutorial menu}       CP/M Learning Tool
T:        {9      Create working diskettes}
T:        {U      Utility menu}
T:        {ESC    Exit to CP/M}
LF:1
T:     {Enter your selection:}
E:
R: = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

First of all, note that the programmer has inserted a line of "=" just above the display, simply to visually separate the display when he is looking at the program listing itself. Then comes a blank R: line, so that the REMARK "Menu displays" may be read more easily. Following another blank REMARK line, we have the actual label indicated by the USE: statement, *MENU1.

Then, the programmer clears the screen so that there will be room for the Main Menu, and the menu appears. Note the two LF: instructions, which insert a blank line in the display so that it's easier to read. After the display of the menu, the subroutine ends. Note the last REMARK line of "=" signs to separate the display from the rest

of the program. The program now returns to the statement after the U: *MENU1 instruction, which is:



```
U:  *STATUS
```

Once again, we'll perform a subroutine before continuing down the main line of the program. In this case, it is the subroutine labeled *STATUS, which looks like this:

```
*STATUS
R:  This routine refreshes the status lines (20-24) on the screen
CUR:0,19
T: - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
T:{CURRENT DRIVE:}$DRIVE:
T:
T:
TNR:
E:
```

The R: instruction tells us what the subroutine does. Note that the CUR: instruction places the cursor on the far left side of the screen (Column 0) 20 lines down. Then, after the highlighting of the CURRENT DRIVE message (by use of brackets), the current drive is given by the use of the string variable $DRIVE, which was set to the actual value of the current drive earlier in the program. Finally, after T: and TNR: (Type with no carriage return) instructions which contain blank spaces to help blank out all the old status information, the subroutine ends. We automatically return to the main program, which continues with the next subroutine, as follows:

```
U:  *CHANGEMSG
```

This subroutine looks like this:

```
R: = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
*CHANGEMSG
R:  This routine simply puts a message in the status area to indicate
    what to
R:  press to change the current drive.
CUR:  15,22
T:  To change the current drive, press "C".
E:
```

and once again, the REMARK explains what the subroutine does. (Careful, just-plain-English documentation within the program listing itself is always a good idea.) Note that the cursor position is defined so that the change message is placed at the right spot on the screen. Then we return to the main program.

Continuing on down the program, we reach the following:

```
*INPUTLOOP3
INMAX:1
CUR:26,17
A:
M:  C ,
JN:*LABEL1
UY:  *CHANGE
J:*INPUTLOOP3
```

This little routine uses another instruction, **INMAX:**, to limit the maximum input allowable to 1 character, and then continue without waiting for a RETURN. This is like an A: instruction that just takes the first key pressed as the answer and moves on. The CUR: statement places the cursor at the correct spot on the screen (after the "Enter your selection" statement), and the ACCEPT and MATCH statements accept the user's response and check if it's the letter "C". If it is a "C," it means the user wishes to change

the correctly logged drive, so the UY: instruction (USE if YES) is activated, and we jump to a subroutine that is a bit too complicated to follow at this point. Let it suffice to say that that subroutine allows us to change the current logged drive, and then returns control to the main program, which loops back through (i.e., jumps back to *INPUTLOOP3) to go through the whole change message again, until the user indicates by some other response that s/he does not want to change the drive.

Then, the program jumps to *LABEL1, which looks like this:

```
*LABEL1
M: 1 , 2 , 3, ,4 , 5 , 6 , 7 , 8 , 9 , U ,
JN:*INPUTLOOP3
CLRS:
M:  1 ,
JY:  *NEWWORD
M:  2 ,
JY:  *SUPERCALC
M:  3 ,
JY:  *CORRECT
M:  4 ,
(etc.)
```

The routine is comparing the user's response, and looking for the number to match, at which point it will JUMP to the new section. For example, if the user typed a "1", it will MATCH the M: instruction above the JY: *NEWWORD statement, and therefore the program will JUMP to the section labeled *NEWWORD. That section will display the message about preparing to run NewWord and take us to NewWord after pausing to allow insertion of the NewWord working diskette.

Any other number, of course, would be matched to its corresponding program. We suggest at this point you sign on the system again, using the CP/M system disk, and run through the first few steps of the Micro Menu, to see once again what the above program accomplishes.

So, you see how the program has accomplished the first few steps of the Main Menu. It is somewhat complicated, but at this point we simply wanted to show that a PILOT program can drive a complicated, interactive system such as the Micro Menus. We hope that, with further practice, you can find similar uses that will fit your needs.

Using PILOT to Run CP/M Programs

The main purpose of a menu system like MICRO.PIL is to link new users to the less-than-friendly operating system, CP/M. Using the PILOT command **CPM:**, you can construct menus and "scripts" that run CP/M commands and programs for you.

A simple example is a menu that formats a diskette if you select *1* from a list of options. An excerpt might look like:

```
M:  1
CPMY:  format b d
```

This line means "Format the diskette in drive B as double-sided." The **format** command is easy to work with, in that it allows you to put parameters like b and d on the command line. Compare this with a strictly interactive program like **sysgen**. If you use the command **CPM: SYSGEN**, the user will be prompted for source and destination drives, just as if he were running sysgen directly from CP/M.

This doesn't make full use of PILOT's automation capabilities, however. If you are always sysgening from a certain drive to a certain other drive, you can feed this information to sysgen directly using a special form of the CPM: command:

```
CPM:  sysgen;a;b
```

Here you are saying "Read the system tracks on disk A and copy them onto disk B." The semicolons stand for the

carriage returns that sysgen expects after drive letters are input.

For the most part, you probably won't be running this sort of utility program from a menu. To run an application program like NewWord, use a command line like this one:

```
CPM:  B:NW!
```

This introduces a new symbol, the **vertical bar**. It tells PILOT to quit reading the PILOT program until the user exits from NewWord. If you leave out the vertical bar, NewWord sees the next PILOT command as input, and strange things will happen.

The XSTATEX File

When PILOT initiates the running of a program like NewWord, it has to clear most of itself out of the computer's memory. When the user is done with NewWord, PILOT reloads itself and picks up where it left off in the .PIL program. So how does PILOT keep track of where it was when it went to sleep?

Whenever PILOT calls a CP/M .COM program, it creates a file called **XSTATEX.PIL** that includes the point in the program where it left off, the locations of all the PILOT program's labels, and the current values of any variables that had been defined. This file is stored on the disk that is currently logged. Whenever PILOT is started, either from scratch or after you exit from a program called by PILOT, it first checks to see whether an XSTATEX file is present. If so, it reads the file and acts according to its contents.

This can occasionally have a curious result. Suppose you're running NewWord from a PILOT menu, and for some reason or other, you're forced to reset your computer. The next time you run the PILOT program, it acts as though you've just exited from NewWord, in effect starting in the middle. We're telling you this in case you've seen this

happen and were confused; it seldom happens and generally clears itself up, since PILOT automatically erases the XSTATEX file after reading it.

Multiple CP/M Commands

It is a simple matter (usually) to string several CP/M commands onto a single CPM: line. For example, this command formats a diskette, copies several files onto it, and reports available space:

```
CPM:  format b d; pip; b:=a:*.com; b:=a:*.pil;; stat b:
```

Note the double semicolons near the end of the line. This is for the special case of using pip in its interactive mode—a carriage return is needed to get out of pip back into the PILOT sequence. Remember that PILOT interprets semicolons as carriage returns. Also, excessively long CPM: lines can cause the computer to lock up, forcing you to reset. Use several CPM: commands in a row if this happens.

The trickiest thing in this whole topic of CPM: comes when interactive programs like NewWord and sysgen are intermingled with other programs on the same CPM: command line. You can get around this by using separate CPM: lines for each program, adhering to the rules covered so far. But if you like to get fancy, examine this example:

```
CPM:  dir a:; dir b:; B:nw! stat a:; sysgen!
```

While obviously nonsensical, this demonstrates how you need to pay attention to the placement of semicolons and vertical bars. This command displays the directories, and then runs NewWord. When the user exits from NewWord, drive statistics for disk A appear and sysgen goes into interactive mode, that is, the user is prompted for drive letters and an exit command.

As a final example, notice the use of double semicolons here:

```
CPM:  sysgen;a;c;;sc!
```

This command runs sysgen without any operator action. The double semicolons feed sysgen the extra carriage return it needs for quitting. Then PILOT runs SuperCalc. If you had used a single semicolon, sysgen would interpret s and c as drive letters being sent to it. If you used a vertical bar in place of the semicolons, the program would work, but the user would have to press RETURN to tell sysgen to quit.

Summary of CPM:

1. Use semicolons to separate programs on the same CPM: line.
2. Use semicolons to separate data being used to simulate keyboard input with interactive programs.
3. Use semicolons to simulate carriage returns with interactive programs.
4. Use vertical bars to allow keyboard input with interactive programs.

6 Some Programming Hints for PILOT

Common Responses

ONE OF THE surest ways to promote computer-frustration from a user is to have too many normal human responses judged as errors. In that light, it is good programming technique to channel the user toward entering information correctly.

One way to accomplish this is to be consistent in your requirements for answers. For example, if there are a lot of yes/no choices in your program, establish the pattern of response in the very first yes/no question, i.e., indicate to the user how the data is to be entered. One way could be as follows:

```
T:  Is this the program you wanted?
T:    Type Y for Yes
T:         N for No
```

This can help establish a pattern, so that fewer data entry errors are made.

Another way to channel the user's responses is to provide multiple choice questions:

```
T:  Which option would you like?
T:    A)  Check Regulations
T:    B)  Compute Tax
T:    C)  Review Personnel Records
T:  Pick an option and type its letter
```

This technique also leads to the building of interactive menus, where the user continues to pick from a choice of several options.

Finally, another way to help cover the normal range of user responses is to include DEF: instructions, where a range of responses is defined as equalling the response needed. For example, you could ask the question:

```
DEF:  $YES  Yes, y , sure , ok , o.k. , Y , yes ,
T:  Is it all right to continue with this program?
A:
M:  $YES
JY:  *CONTINUE
```

In this example, any of the responses listed in the DEF: instruction would allow the MATCH instruction to consider that the question had been answered in the affirmative, and the program would continue.

Matching—Various Instructions and Techniques

MATCH statements are very important in PILOT programs. They allow the computer to inspect and compare a user's response, and make a decision based upon that response. Therefore, you have to be careful that you match the answers you want to, and don't match the answers you don't want to. There is the obvious solution of only allowing an exact, character-by-character match, but it is likely that, as you write more complex programs, you will find that matching a pattern will be more useful.

The first thing to understand is how a MATCH instruction works with commas and spaces to interpret a pattern. Commas are considered to separate different elements from each other, and spaces are used to define how the individual elements are matched.

The first pattern we'll examine is one where an exact match is required for each element in the pattern. For example, let's suppose that we want only the single-digit numbers 1, 2, and 3 to be valid in response to a question. In this case, we must surround each individual element of the pattern with spaces, to indicate that no character is acceptable either before or after the desired character. We

would type the MATCH instruction as follows (pay close attention to the spaces):

```
M:  1 , 2 , 3 ,
```

This statement would match only the characters 1, 2, or 3, and an entry of 12 or 21 or 43 would not be matched (i.e., would cause an "N" condition to be true after the MATCH statement). The spaces before and after each digit indicate that it must be a single digit only.

What if we wanted to match only the first letters of words, for example? In that case, we would place a space in front of each element:

```
M:  A, B, C,
```

This instruction would match any word beginning with an A, B, or C. The comma without a space after each element indicates that any characters after the first ones would be acceptable. "APPLES", "B-52'S", and just plain "C" are fine; lower case letters would also match.

Next, we could allow any word to match two *ending* letters by not putting any spaces before each element, but putting spaces afterwards, as follows:

```
M:it ,en ,or ,
```

This statement would match any word ending in "it", "en", or "or".

Finally, we could allow any entry which contained one of the desired elements in it to match. For example:

```
T:  Give me a word with a vowel in it.
A:
M:a,e,i,o,u,
TY:  RIGHT!
TN:  I'll bet your word has the letter "Y" in it.
```

In this instance, any word with one of the five normal vowels would cause a match, and thus get the RIGHT! response.

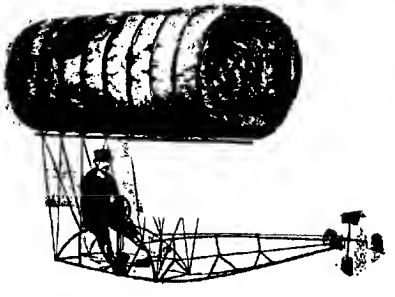
One other consideration when matching patterns is the DEF: statement talked about above. The DEFINE instruction can be used to extend the number of matches allowed, by allowing any of several responses to satisfy the requirements of a MATCH.

There are two additional MATCH instructions which you may find useful. The first, **MC:** (Match including commas), is useful when you are trying to match text which contains commas. Since the comma is normally used to separate the elements of the match, a user-entered comma would confuse the match. In that instance, if there is a chance that a comma might be entered, use the instruction MC: and enter the character "^" as the separator between the elements of the match.

Finally, one more useful MATCH instruction is the **MY:** (MATCH if YES) instruction. This instruction allows the progressive testing of responses, so that a series of matches can be required for any given element. For example, if you wanted to accept a nearly correct answer from the user, you could write:

```
*TRYAGAIN
T:  What is the largest river in the U.S.A.?
A:
M:  M,
TN:  Sorry, that's not right.
JN:  *TRYAGAIN
MY:  Mis,
MY:ip,
TN:  Sorry, that's not right.
JN:  *TRYAGAIN
TY:  Right!  The Mississippi!  Did you spell it exactly
TY:    right?
```

In this case, the program first checks to see if the answer starts with "M". If it doesn't, it asks the question again. If the answer does start with "M", the program then checks if it starts with "Mis," and if it contains an "ip" in it. If it does, the user has almost certainly answered correctly, even if s/he had misspelled the word slightly. In this way, progressive MY: statements can be used to allow for inexact spelling.



7 PILOT Instructions—Explanations

IN THIS SECTION, you will find all of the instructions in the PILOT program in alphabetical order, and information on how to use them. Also included are definitions of "labels," "conditionals," and "special characters."

A: Accept Answer

This instruction tells the computer to wait indefinitely for the user's response. If nothing is typed after the colon, the response will only be saved temporarily for later matching. If a numeric or string variable is added after the colon, the user's response will be stored under that variable name, and may be used when desired later in the program.

Examples:

```
T:  Which option would you like, 1 or 2?
A:
M:  1
TY:  Okay, here's option #1;
```

```
T:  What's your name?
A:  $NAME
T:  Hi, $NAME
```

BELL: Alert User

This instruction tells the program to beep the terminal's buzzer or beeper. It can be used to alert the user of a special condition. Conditionals are valid, so that BELLY:, for example, will cause the buzzer to ring only if the preceding statement matched the response in a MATCH instruction.

Example:

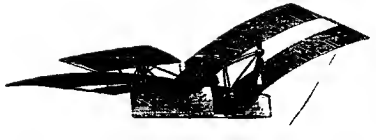
```
T:  What is 1 + 1?
A:
M:  2
BELLY:
TY:  RIGHT!
```

C: Compute

The COMPUTE instruction is used to get the computer to perform mathematical computations. Only addition and subtraction may be done, and the numbers must be whole integers and the result must be less than 32,567 and greater than −32,568. Operations will be performed from left to right. Parentheses will *not* determine precedence, which would be superfluous with addition and subtraction.

Examples:

Expression	*Operation*
C: X = 2	Store the value "2" in the variable "X".
C: X = X + 1	Add 1 to the value already stored in "X".
C: TOTAL = X + Y −17	Add the value stored in the variable "X" to the value stored in the variable "Y", subtract 17 from the subtotal, and store the result in the variable named "TOTAL".
C: PRICE = COST + TAX	Add the value stored in the variable COST to the value stored in the variable TAX, and store the result in the variable PRICE.



NOTE: When numeric variables are used in COMPUTE instructions, they don't have to be preceded by a "#" sign. However, in other lines like T:, they must have the # directly in front of the variable name, e.g., #AMTDUE.

CASE: Determine Action in Each Case

This instruction tells the computer to take action based on the value of a numeric variable.

Example:

```
T:  Which option do you want?
T:  1) Parts  2) Service  3) Administration
A:  #OPTION
CASE (#OPTION):  *PARTS, *SERVICE, *ADMIN
```

In this example, if the option selected by the user is 1, the numeric variable #OPTION will contain the value "1", and PILOT will jump to the label *PARTS.

CH: Chain

This instruction tells the program to leave the current PILOT program and go to another one (the other program must be on the logged-in disk drive).

Example:

```
T:  Which program would you like now?
T:   A) Reference  B) Tutorial
A:
M:  A
CHY:  REF.PIL
```

If the user typed "A", the program would go to the REF.PIL program, and run it. The computer will *not* return automatically to the old program, unless told to do so with another CH: command at the end of REF.PIL.

CLRS: Clear Screen

This instruction completely blanks the screen, and leaves the cursor at the home position at the upper left-hand corner of the screen.

Example:

```
T:  Would you like to start over?
A:
M:  YES
JN:  *NEXT
CLRS:
JY:  *START
```

Conditionals

A "conditional" is the part of a PILOT instruction following the name of the instruction and before the colon. The conditional tells the instruction to proceed if the last match succeeded or didn't succeed.

There are several ways a conditional may be constructed. First, a conditional may be either a "Y" or an "N", for yes or no.

Example Y/N:

```
T:  Would you like option 1? YES or NO
A:
M:  NO
TY:  So, you don't want to use option 1.
TN:  Here's option 1:
```

In the above example, note how the two negative situations are used to equal one positive.

A conditional may also be a numeric variable, which will be considered as a match if its value is greater than zero, or as a "no match" if its value is equal to or less than zero. When used in this manner, a conditional can be used to distinguish how many times a subroutine has occurred, or whether a user has answered with a positive reply.

Example Numeric Variable:

```
T:  Would you like one of the following options?
T:   1) Parts      2) Sales     3) Service
T:   Type a number, or "0" (zero) if you wish to
T:   continue to the next step.
A:  #OPTION
U(#OPTION):  *SELECTOPTION
T:  The next step is:
(etc.)
```

CPM: Execute a CP/M Command

This instruction will automatically run another CP/M program and then return to the PILOT text when the program is ended.

Example:

```
T:  Which CP/M program would you like to run?
T:  A) Stat  B)Pip
A:
M:  A
CPMY:  STAT
M:  B
CPMY:  PIP
```

A CPM: instruction may also contain a "¦" (vertical line) sign after the program name. This is for use with CP/M transient commands that require operator input from the keyboard. PIP.COM, for example, may be called by CPM: so that it prompts the user for files to copy:

```
CPM: PIP¦
```

Programs and arguments may be strung together on a CPM: command line. Semicolons are used where carriage returns would normally occur in the flow of running the programs outside of PILOT. Frequently a program name and its arguments are separated by semicolons; programs are separated from other programs by two semicolons.

Example:

```
CPM: PIP¦A:=B:file1¦A:=B:file2¦¦STAT A:
```

CUR: Set Cursor

This instruction sets the cursor position. First type the column number, and then the line number. The line and column numbering starts with zero, i.e., the coordinates of the upper lefthand corner are 0,0.

Example:

```
CUR: 3,13
```

would set the cursor in the 4th column, on the 14th line.

DEF: Define the Value of a String Variable

This instruction will define a String Variable as being equal to any one of the given responses.

Example:

```
DEF:  $JOHN  John , Johannes , Jon , Jean , Jack
```

If the variable $JOHN was later matched to a response, any one of the names above would count as a match.

DI: Disable ESCAPE Key

This instruction disables the ESCAPE key, so that if that key is accidentally pressed, PILOT will not exit to CP/M.

Example:

```
*BEGIN
R:  Escape key will not cause exit to CP/M
DI:
```

E: End Subroutine

This instruction signals the end of a subroutine, and will return the program to the statement after the U: instruction which took it into the subroutine.

Example:

```
UY:  *MESSAGE
J:  *QUESTION

*MESSAGE
T:  Please answer the question again.
E:
```

After going to the subroutine *MESSAGE, the program would type the statement, end the subroutine, and return to the next statement after the U: instruction, in this case the J: *QUESTION instruction.

NOTE: If E: is used where no subroutine is pending, the PILOT program will end and control will be returned to the CP/M operating system, i.e., the E: instruction will function in the same way as the END: instruction (see END:).

EI: Enable ESCAPE Key

This instruction enables the ESCAPE key to function after it has been disabled by the DI: command.

Example:

```
R:  User might need ESC key after this point
EI:
```

END: End PILOT Program

This instruction ends the PILOT program immediately and returns control to the CP/M operating system.

Example:

```
T:  Are you finished with this program?
A:
M:  YES
JN:  *START
END:
```

ERASTR: Erase String Variable

This instruction erases all characters stored in all string variables so that they can be used over again. Conditionals may be used.

Example:

```
T:  What's your answer?
*ANSWER
A:  $ANSWER
M:  $OK
TN:  Sorry, I couldn't understand that.  Could
TN:  you please try again?
ERASTRN:
JN:  *ANSWER
T:  Okay, I understand.
```

ESC: Define Escape Sequence

This instruction enables PILOT to jump to a subroutine if the ESCAPE key is hit during an ACCEPT or WAIT instruction. It must be the first instruction in the program, except for REMARK instructions.

Example:

```
R:  First, an escape subroutine
ESC:  *ESCMESSAGE
*START
*ESCMESSAGE
T:  Are you sure you want to get out of PILOT?
A:
M:  YES
ENDY:
```

EXIST: Check Existence of CPM Program

This instruction will cause the program to look at the disk directory and note whether a certain file exists on the disk. You must include ".COM" in .COM filenames submitted to EXIST:. The outcome of EXIST: automatically sets up a conditional, as shown in the example.

Example:

```
T:  Which program would you like to run?
A:  $PROGRAM
EXIST:  $PROGRAM
TN:  Sorry, that program is not on the logged-in
TN:  disk drive.
```

HOLD: Hold Scroll

This instruction causes the screen display to stop or hold until the user presses ENTER or RETURN. The user may also return to a specified label by typing "R". This allows the user to review material if it has scrolled off the screen. HOLD may be used with conditionals.

Example:

```
*READAGAIN
T:  The projection model will assume the
T:    following:
T:   (etc.)
```

and then later in the program:

```
T:  Do you remember the projection model
T:   assumptions?
T:  If you would like to review them, press "R".
T:  Otherwise press RETURN to continue.
HOLD:  *READAGAIN
```

NOTE: The label given after the colon in the "HOLD:" instruction could also reference a routine farther down the program. However, the HOLD instruction is especially useful for going backwards in the program to allow the user to review material.

INMAX: Set Input Line Length

This instruction determines the maximum number of input characters allowed in a string variable. When that number of characters is reached, the computer will automatically go to the next instruction.

Example:

```
T:  Which option do you want?  1, 2, or 3
INMAX: 1
A:
M: 1
JY:  *OPTION1
 (etc.)
```

J: Jump

This instruction tells the computer to JUMP from the current spot to the label specified. The program will not jump back unless told to do so with another JUMP instruction.

Example:

```
T:  Which option would you like, A or B?
A:
M:  A
JY:  *OPTIONA
```

Labels

A label gives a name to a section or instruction of a PILOT program. The label may then be referenced by another PILOT instruction in another part of the program, thus directing the flow of the program to the label.

Labels must begin with an asterisk (*), and be either on a line by themselves, or the first statement on a line.

LF: Line Feed

This instruction tells the program to insert the indicated number of blank lines onto the screen.

Example:

```
T:  Right!
LF:  4
T:  The next question is:
 (etc.)
```

M: Match
MC: Match Including Commas

These instructions tell the computer to compare a user's answer to the answer given after the colon. For a complete description, refer to Section 6.

OUT: Output to an I/O Port

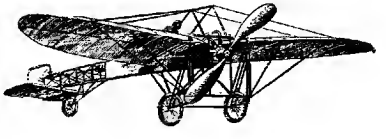
This instruction tells the computer to output a byte to any I/O port to control external devices. The byte to put out follows the colon, then a comma, then the port. Both numbers must be two digits and must be represented in hex.

Example:

```
T:  Do you want to send that byte out?
A:
M:  YES
OUTY:  02,C0
```

PR: Print

This instruction tells the computer to print the text following the colon on the currently assigned CP/M list device.



Example:

```
PR:  This line would show up on the printer
```

R: Remark

This instruction indicates that the text following the colon is a remark, and will not be printed on the screen when the PILOT program is run.

Example:

```
R:  This text would not show up on the screen
R:  when the PILOT program containing it is run.
```

RESET: Reset Numeric Variables to Zero

This instruction sets the value of all numeric variables in the program to zero so that they may be used over again.

Example:

```
T:  Do you want to compute some more totals?
A:
M:  YES
RESETY:
```

NOTE: This instruction will only work if the PILOT version you are using is above Version 5.1 (i.e., the instruction does not work in version 5.1).

SAVE: Save Input Text

This instruction stores the last keyboard response into the string variable you specify. The saved response may then be used in another part of the program.

Example:

```
T:  What's the title of that book you liked?
SAVE:  $BOOK
```

and then later in the program:

```
T:  You mentioned that you liked the book named:
T:      $BOOK.  What other things do you like?
```

Special Characters

The two special characters left brace "{" and right brace "}" will be interpreted by PILOT as blank spaces, but the left brace will turn highlighting on, and the right brace will turn highlighting off.

The special character "$" represents a string variable, while "#" precedes a numeric variable. Whenever you want to display either of these characters, you must use two of them in sequence to produce one.

T: Type Text

This instruction types the text following the colon onto the screen.

Example:

```
T:  This text will appear to the user
```

TNR: Type Text With No RETURN

This instruction types the user's response onto the screen on the same line as the statement.

Example:

```
TNR:  Which option would you like?  #
```

In this example, the user would enter his answer right after the "#" sign.

U: Use

This instruction tells the computer to go to the labeled subroutine, perform the instructions there until it reaches an E: instruction, and then return to the main flow of the program (i.e., the next instruction after the U: instruction).

Example:

```
*BEGIN
T:  How much is 2 + 2?
A:
M:  4 , four
TY:  Right!
UN:  *ERROR
etc.

*ERROR
T:  Sorry, wrong answer.  Try another.
E:
```

WAIT: Accept Answer (Timed)

This instruction will normally give the user six seconds to start typing an answer to a question. If the user does not start typing by the end of six seconds, the computer will continue as if the user had typed the word "TIMEOUT". "TIMEOUT" can then be used in an M: instruction.

Example:

```
T:  What's your name?
WAIT:
M:  TIMEOUT
TY:  Don't you know your own name?
```

If a string variable is after the colon and the user does not begin typing in time, then the string "TIMEOUT" is assigned to the variable. If a numeric variable is after the colon, and the user does not begin typing in time, then the value "0" (zero) is assigned to that variable, and may be used as a conditional in future statements.

Example:

```
T:  How old are you?
WAIT:  #AGE
J (AGE):  *NAME
T:   What's  the matter?   Afraid of  me?   You
T:   didn't even start typing before six whole
T:   seconds were up.
T:   Try again
J:  *START

*NAME
T:  So, you're #AGE years old.
T:  What's your name?
WAIT:  $NAME
M:  TIMEOUT
TY:  Don't you know your own name?
TN:  Hi, $NAME.
```

Finally, the length of time the computer waits for a response can be adjusted by making the program loop through the wait routine several times.

Example:

```
T:  Question?
C:  W=3

*ANSWERWAIT
WAIT:  $ANSWER
M:  TIMEOUT
JN:  *NEXT
C:  W=W-1
J(W):  *ANSWERWAIT
T:  Time has expired.

*NEXT
T:  Thank you for your answer.
```

In this example, we have instructed the computer to go back through the routine three times, which would give the user eighteen seconds to start typing.

WAIT can also be used to display a message for a certain time before proceeding to the next instruction.

8 Error Messages

Message	*Problem and Action*
(FILENAME): CAN'T ACCESS	A file could not be opened. Check that you spelled the program name exactly the same as the name on the diskette. Check that the proper diskette is in the drive, and that you've included the drive letter if needed.
(INSTRUCTION): UNRECOGNIZED INSTRUCTION	A non-PILOT command was given in the program. Check that the colon is placed immediately next to the instruction and that the instruction is a legal PILOT command.
(LABELNAME): LABEL NOT FOUND	A JUMP instruction was given to a non-existent label. Check to be sure that the label exists, that it is spelled exactly the same as in the J: command, and that there are no spaces between the asterisk and the label name.
MISSING LABEL	PILOT encountered a JUMP command which did not have a label name after it. Enter the label name after J:. If the label exists, check to be sure there are no spaces between the asterisk and the label name.
STACK OVERFLOW	Subroutines are nested too deeply. Since PILOT allows up to 512 levels of subroutines, this message is likely to occur only in the event of a runaway recursive subroutine, i.e., one that is calling itself.

Appendix—Summary of PILOT Instructions

THE following appendix gives the syntax of every PILOT instruction. Any item in brackets [...] is an optional item. The word "comments" after the colon means that text may be placed in that position, but will simply be ignored when the program is run. Thus, the space after the colon may be used to document the program listing.

Instruction	Syntax
A:	Accept Answer
	[label] A [cond] : [label] A [cond] : $string variable [label] A [cond] : #numeric variable
BELL:	Alert User
	[label] BELL [cond] :
C:	Compute
	[label] C [cond] : num variable = expression
CASE:	Determine action in each Case
	[label] CASE (num variable) : label, label, (etc.)
CH:	Chain
	[label] CH [cond] : program-name
CLRS:	Clear Screen
	[label] CLRS [cond] :
CPM:	Run CPM Program
	[label] CPM [cond] : command[;arguments;; command;;etc.] [label] CPM [cond] : command[!accept keyboard input] [label] CPM [cond] : ....$string var... #numeric var...

CUR: Set Cursor

[label] CUR [cond] : column,row

DEF: Define a Variable

[label] DEF [cond] : $variable string

DI: Disable ESCAPE key

[label] DI [cond]: [comments]

E: End Subroutine; Return to Main Program

[label] E [cond] : [comments]

EI: Enable ESCAPE key

[label] EI [cond]: [comments]

END: End PILOT program; Return to CPM

[label] END [cond] : [comments]

ERASTR: Erase String Variable

[label] ERASTR [cond] :

ESC: Define Escape Sequence

[label] ESC [cond] : [*] label

EXIST: Check existence of CPM Program

[label] EXIST [cond] : program name

HOLD: Hold Scroll

[label] HOLD [cond] : [*] label

INMAX: Set Input Line Length

[label] INMAX [cond] : integer
[label] INMAX [cond] : [*] numeric var

J: Jump

[label] J [cond] : [*] label

LF: Line Feed

[label] LF [cond] : decimal number

M: Match

[label] M [cond] : $string variable
[label] M [cond] : pattern[,pattern...,etc.

MC: Match including commas

[label] MC [cond] : pattern[,pattern...,etc.

OUT: Output to an I/O port

[label] OUT [cond] : byte,port

PR: Print

[label] PR [cond] : text

R: Remark

[label] R : comments

RESET: Reset Numeric variables to zero

[label] RESET [cond] : comments

SAVE: Save input text

[label] SAVE [cond] : $string variable

T: Type text

[label] T [cond] : text

TNR: Type text with no RETURN

[label] TNR [cond] : text

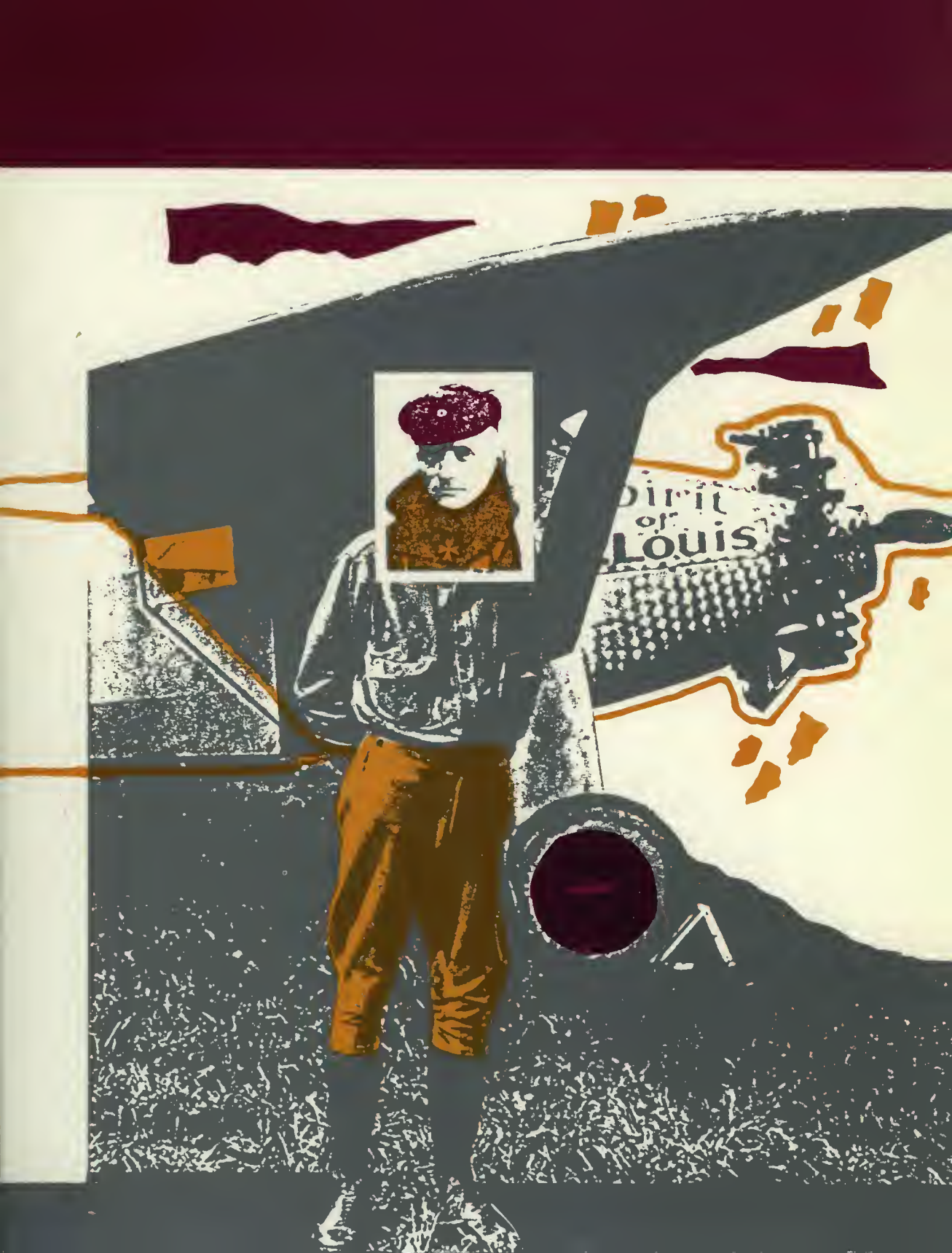
U: Use

[label] U [cond] : [*] label

WAIT: Accept Answer (Timed)

[label] WAIT [cond] : $string variable
[label] WAIT [cond] : #numeric variable
[label] WAIT [cond] :

PILOT User's Guide
Index


Spirit
of
Louis